Define a class to store an extensible array. It should be possible to index into the array, and it should be possible to extend the array using a push_back operation. A skeleton definition of the class is given below. You are to fill in the ... portions, and then type the entire struct into the textbox of the assignment. You should implement the protocol that at any time there must be exactly one pointer to space allocated on the heap, and your implementation must prevent memory leaks and dangling pointers.

```cpp
struct eArray{
    private:
        ...
    public:
        eArray(...) // construct an array with 0 elements
        int &operator[(int i){...}
        // return a reference to the ith element of the array
        void push_back(int v){...}
        // Append v to the current array
        // Use a simple implementation: allocate a new array to
        // accommodate the extra element v. Then copy the current
        // array into it. Copy v, and delete the current array.
        int size() const {...}
        // return the current size of the array
        // "const" says this function will not change the receiver
        eArray(const eArray &rhs){...}
        // copy constructor
        ~eArray(){...}
        // destructor
    }
}
```

Due on 2020-04-08, 23:59 IST
eArray& operator=(const eArray &rhs){...}  
// assignment operator
}

Using this it should be possible to write programs of the following kind. The following programs has already been typed in and will be used to test your code.

void f(eArray A){A[5] = 5000;}  // changes only local copy
void g(eArray &A){A[6] = 6000;}  // changes original

int main(){
eArray A,B;
int n; cin >> n;
for(int i=0; i<10; i++){
    int v; cin >> v;
    A.push_back(v);
}
// at this point A should contain all values read.
f(A);
g(A);
B = A;
B[8] = 800;
A[8] = 8000;
for(int i=0; i<B.size(); i++) cout << B[i] << ' ';  
cout << endl;
}

On input 10 0 10 20 30 40 50 60 70 80 90 this should print 0, 10, 20, 30, 40, 50, 6000, 70, 800, 90.

Note that the eArray as defined above is similar to the vector class in the standard library. Your implementation can be very simple; whenever an element is appended, you can just allocate a new array of a larger size, copy the elements, and delete the array. In the vector class, a large memory is allocated every time the current allocation is found inadequate; this way the number of allocation operations is reduced.

### Sample Test Cases

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Case 1</strong></td>
<td>0 10 20 30 40 50 60 70 80 9</td>
<td>0 10 20 30 40 50 6000 70 800 90</td>
</tr>
<tr>
<td><strong>Case 2</strong></td>
<td>1000 2000 3000 4000 5000 60 00 7000 8000 9000 10000</td>
<td>1000 2000 3000 4000 50 00 6000 6000 8000 800 10000</td>
</tr>
<tr>
<td><strong>Case 3</strong></td>
<td>4754 8655 2218 29375 30 792 6705 8834 9720 9362 3633</td>
<td>4754 8655 2218 29375 3 0792 6705 6000 9720 80 0 3633</td>
</tr>
</tbody>
</table>
The due date for submitting this assignment has passed.
As per our records you have not submitted this assignment.
Sample solutions (Provided by instructor)

```cpp
#include <iostream>
define repeat(x) for(int _iterator_i = 0, _iterator_limit = x; _iterator_i < _iterator_limit; 
#include <cmath>
define main_program int main()
#define repeat(x) for(int _iterator_i = 0, _iterator_limit = x; _iterator_i < _iterator_limit; 
struct eArray{
private:
text elements;
text n;
public:
eArray(int n=0; elements = NULL; } // construct an array with 0 element
int operator[](int i){ return elements[i];}
// return a reference to the ith element of the array
void push_back(int v){
  int* newE = new int[n+1];
  for(int i=0; i<n; i++) newE[i] = elements[i];
delete[] elements;
elements = newE;
elements[n] = v;
++n;
}
int size() const {return n;}
eArray(const eArray &rhs){
  n = rhs.n;
elements = new int[n];
  for(int i=0; i<n; i++) elements[i] = rhs.elements[i];
}
operator=(const eArray &rhs){
  if(this == &rhs) return *this;
  // If it is a self assignment, we do nothing.
delete[] elements; // Release the memory allocated earlier
  n = rhs.size();
elements = new int[n];
  for(int i=0; i<n; i++) elements[i] = rhs.elements[i];
  return *this;
}
)
};
eArray& operator=(const eArray &rhs){
  if(this == &rhs) return *this;
  // If it is a self assignment, we do nothing.
delete[] elements; // Release the memory allocated earlier
  n = rhs.size();
elements = new int[n];
  for(int i=0; i<n; i++) elements[i] = rhs.elements[i];
  return *this;
}
};
void f(eArray A){A[5] = 5000;} // changes only local copy
void g(eArray &A){A[6] = 6000;} // changes original
int main(){
eArray A,B;
for(int i=0; i<10; i++){
  int v; cin >> v;
  A.push back(v);
}
```
p_ - f(A);
g(A);
B = A;
B[8] = 800;
A[8] = 8000;
for(int i=0; i< B.size(); i++) cout << B[i] << " ";
cout << endl;
}